

CS 350 Operating Systems, Spring 2024
Programming Project 2 (PROJ2)
Process scheduling in xv6 - Q&A

Out: 02/26/2024, MON

Due date: 03/20/24, WED 23:59:59

Group 9 Members: Anita Hong, Andrew Ferreira, Chao Lin, Jiaming Chen

Q1: (10 points) Does the xv6 kernel use cooperative approach or non-cooperative approach to gain control while a user process is running? Explain how xv6's approach works using xv6's code.

The xv6 kernel uses a non-cooperative approach to gain control while a user process is running. We can see this in action by taking a look at how interrupts and scheduling work:

- The kernel relies on timer interrupts generated by the hardware periodically. When an interrupt occurs, the CPU's context is saved in a trapframe and control switches to the interrupt handler code in kernel/trap.c.
- If the interrupted process was a user program, the kernel can then decide to schedule a new process using the scheduler() function defined in kernel/proc.c. This shows that control passes to the kernel without user permission through preemption.
- To actually perform the context switch, the old process state is saved and a new one is loaded. So the kernel is fully in charge of scheduling.

In a cooperative approach, the processes voluntarily yield control back to the kernel when they're ready to switch. Nowhere in the code do we see user processes explicitly yielding control or blocking the kernel. Control always flows through interrupt handling to the kernel scheduling logic.

By using timer interrupts to preempt processes and its own in-kernel scheduling functions, xv6 exhibits a classic non-cooperative multitasking approach. The kernel can regain CPU control at any time without user process cooperation. Therefore, the xv6 kernel uses a non-cooperative approach

Q2: (10 points) After fork() is called, why does the parent process run before the child process in most of the cases? In what scenario will the child process run before the parent process after fork()?

- After a process calls fork(), it creates a new child process. This is like making a copy of itself. Now both the parent and child are ready to run.
- In xv6, processes take turns running using a "round robin" system. This means each process only gets to run for a little while, then it's the next one's turn.
- Usually, the parent process was running first before the fork(). So after the fork(), it gets to finish its turn before the child starts. That's why normally the parent runs first.
- But there's a small chance that the parent's turn ends right after the fork(). Then the kernel will let the child have its turn immediately.
- We can also make the child go first on purpose. There's a function called yield() that tells the kernel "I'm done, let someone else have a turn now." If we call yield() inside fork(), the parent gives up its turn, so the child gets to run first.

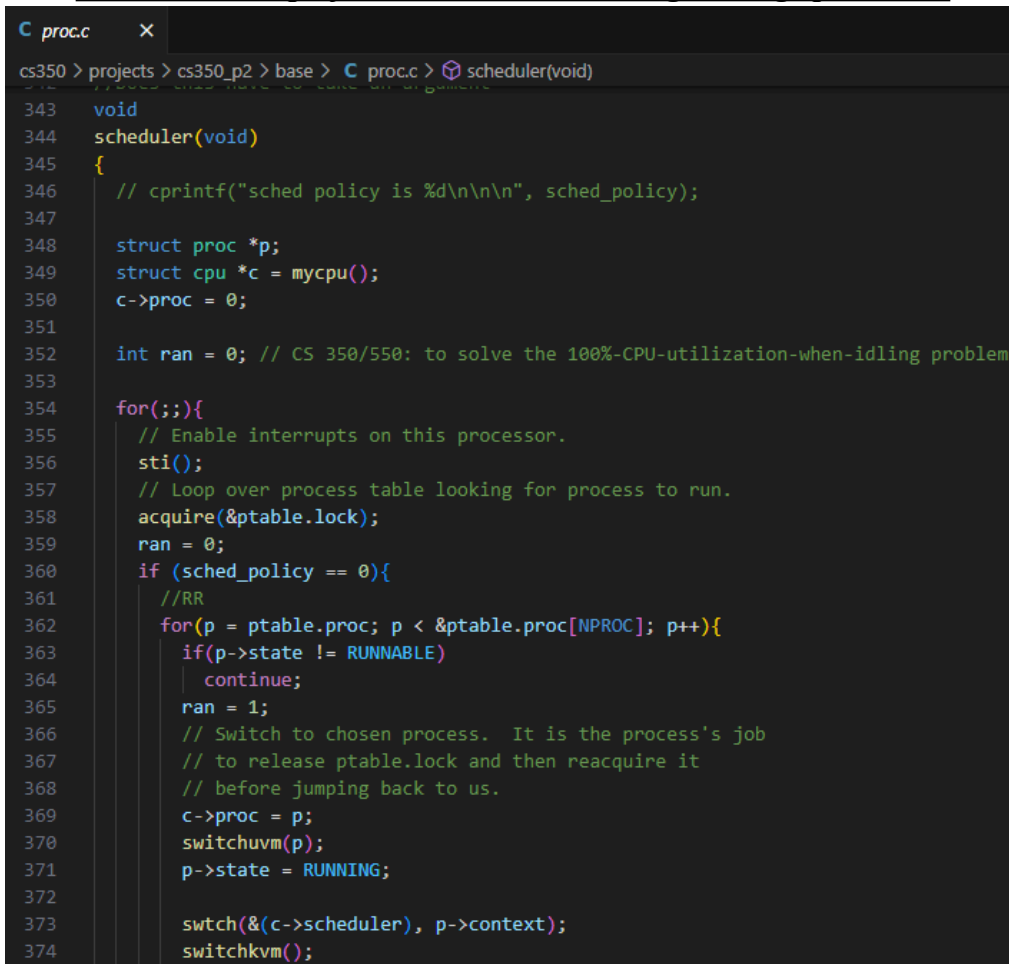
Q3: (10 points) When the scheduler de-schedules an old process and schedules a new process, it saves the context (i.e., the CPU registers) of the old process and load the context of the new process. Show the code which performs these context saving/loading operations. Show how this piece of code is reached when saving the old process's and loading the new process's context.

The key functions involved in this context switching process are:

1. switchvm() Prepares the new process's kernel stack and sets up the kernel's page table.
2. swtch() Saves the current context and switches to the new process's context.
3. switchkvm() Switches back to the kernel's page table and releases the process table lock.

The code that performs these context saving/loading operations is found in the scheduler function in proc.c.

The code which performs these context saving/loading operations:



```

C proc.c x
cs350 > projects > cs350_p2 > base > C proc.c > scheduler(void)
343 void
344 scheduler(void)
345 {
346     // cprintf("sched policy is %d\n\n", sched_policy);
347
348     struct proc *p;
349     struct cpu *c = mycpu();
350     c->proc = 0;
351
352     int ran = 0; // CS 350/550: to solve the 100%-CPU-utilization-when-idling problem
353
354     for(;;){
355         // Enable interrupts on this processor.
356         sti();
357         // Loop over process table looking for process to run.
358         acquire(&ptable.lock);
359         ran = 0;
360         if (sched_policy == 0){
361             //RR
362             for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
363                 if(p->state != RUNNABLE)
364                     continue;
365                 ran = 1;
366                 // Switch to chosen process. It is the process's job
367                 // to release ptable.lock and then reacquire it
368                 // before jumping back to us.
369                 c->proc = p;
370                 switchvm(p);
371                 p->state = RUNNING;
372
373                 swtch(&(c->scheduler), p->context);
374                 switchkvm();

```

How this piece of code is reached when saving the old process's and loading the new process' context:

1. When the scheduler determines that a new process should run, it sets the state of the old process to RUNNABLE and calculates its pass value.
2. The old process's context is saved when it is about to be de-scheduled, which happens in the sched() function.
3. The scheduler then selects the new process that should run next and marks it as 'RUNNING'.
4. The context of the new process is loaded when the scheduler calls 'swtch()' with the new process's context as an argument.